# Eiron
# High-level Design, Roadmap, and Status

Oskar Lappi

November, 2023

# What is Eiron?

Very simple 2D Monte Carlo neutral particle transport solver

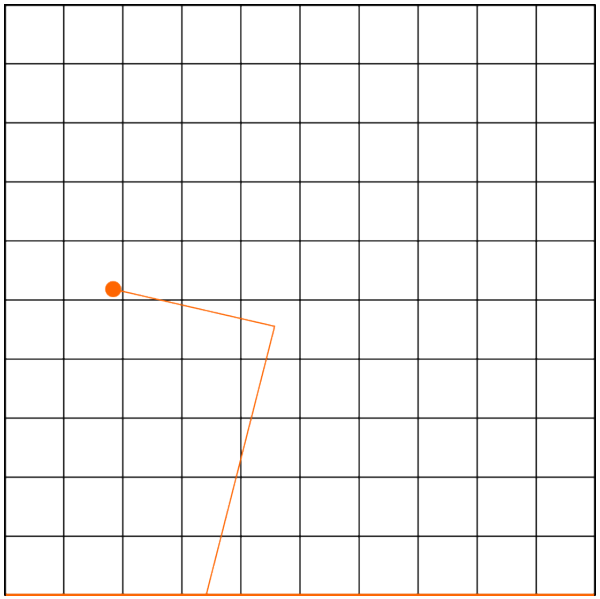
Goal design and benchmark parallel architectures using Eiron, apply lessons learned to EIRENE

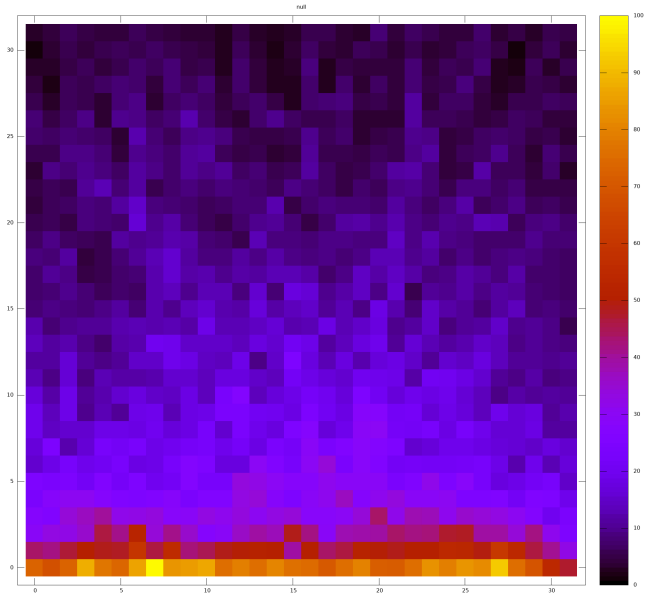
# Presentation outline

- Eiron design
  - Model overview: simulation, estimation
  - System components
  - Parallel designs: pros and cons
- Domain-decomposition challenges and solutions
- Charon: library for asynchronous domain-decomposed communication patterns
- Eiron roadmap and status

# Model overview

# Model overview





## *Simulate* particle sample

- IN: particle sources, background fields (2D grids)
- OUT: particle trajectories (point paths)

## *Estimate* fields ("tally")

- IN: particle trajectories, background fields (2D grids)
- OUT: estimate fields (2D grids)

# Simulation: algorithm

**Input** a particle position, velocity, and species

1. Generate a random variable $longevity \sim Exp(1)$
   - The *longevity* is the length of the step if the MFP $= 1$
2. Select the *collision context* by the particles species
   - *Collision context* = grid of collision rates for a particle species
3. Follow the particle's heading
   Integrate over collision rates until the integral = *longevity*
   Where the integral ends is the end-point of the step
4. Sample collision event from collision rates in end-point cell
5. Update particle position, velocity and species from event
6. If the collision event keeps the particle alive, repeat

# Simulation: core ideas

**Deterministic multi-threaded Monte Carlo**:
Regardless of number of threads, for a given seed, each particle is simulated using the same sequence of random numbers

Goal Simulation should be completely deterministic

Crucial for comparing different parallel designs

**Physics parameters in config** including chemical species
no hardcoded hydrogen/photon cases in source

Goal Separate software and domain science concerns

# Simulation: limitations

- Grid repr: 2D rectangular grid of square cells, **no other kind**
- Wall geometry: $=$ the grid's boundaries, **no polygons**
- Collision rate: **not dependent on any test particle property**

# Estimation

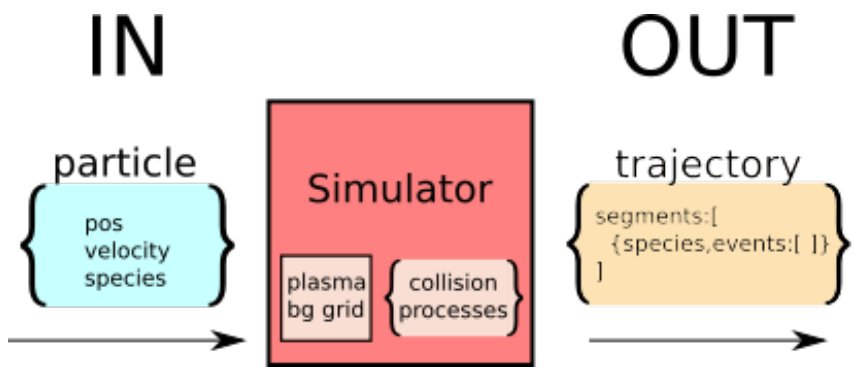
Estimation is currently done using track-length in cell

## Estimation: limitations

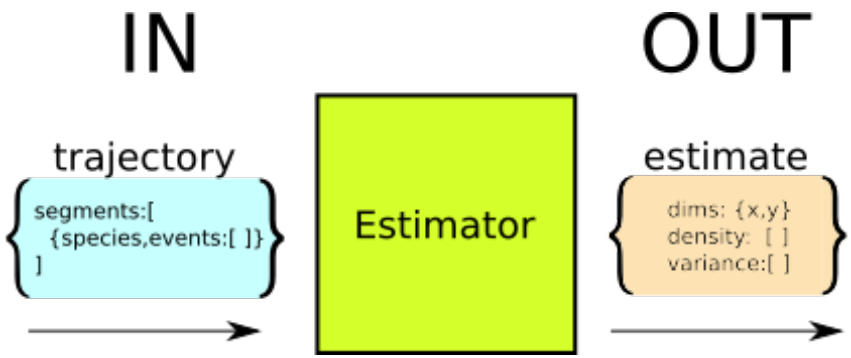
- Variances not calculated yet, will be tackled once domain decomposition is implemented

# System components

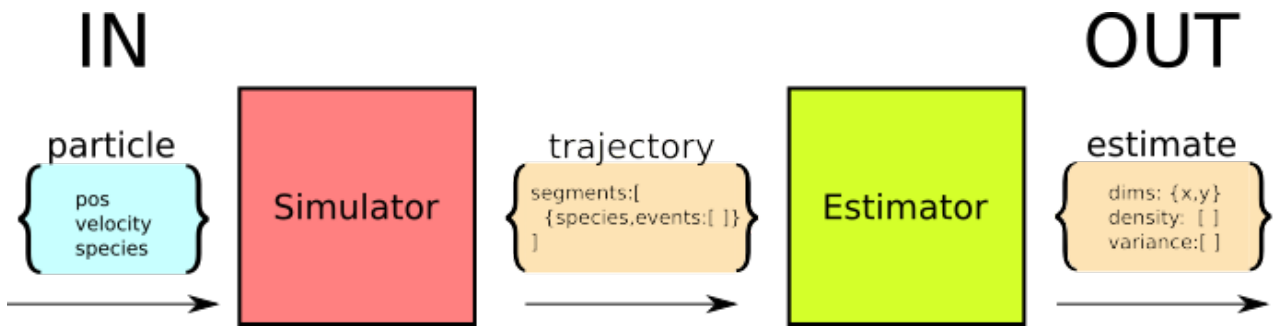
# System components: Simulator



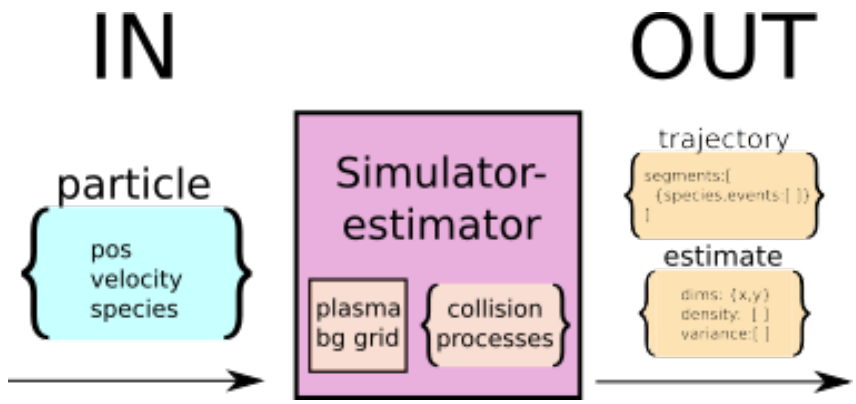
# System components: Estimator



# System components: Simulator and estimator pipeline



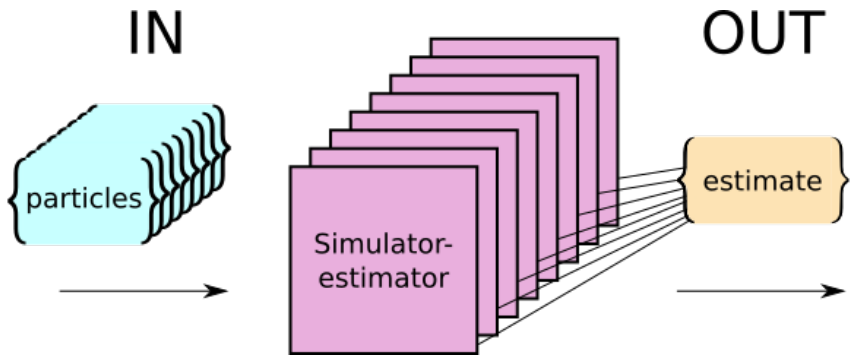
# System components: Simulator-estimator



# Parallel designs

# Parallel designs: OMP shared grid

## Equivalent to current design of EIRENE+OMP

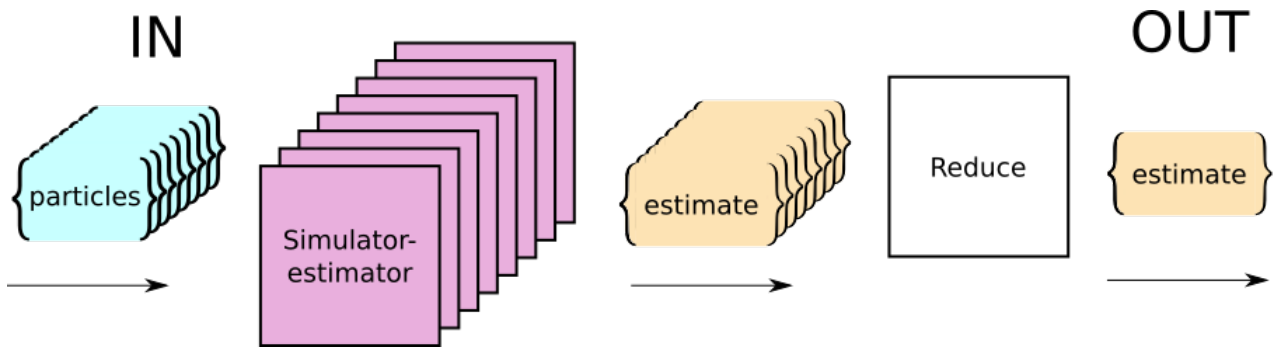



+ Grid's memory footprint is constant
- Shared mutable state → RW sync severely limits scalability
- Can only use as many workers as there are on one node
- Whole grid must fit on one node

STATUS: DONE

# Parallel designs: OMP private grid

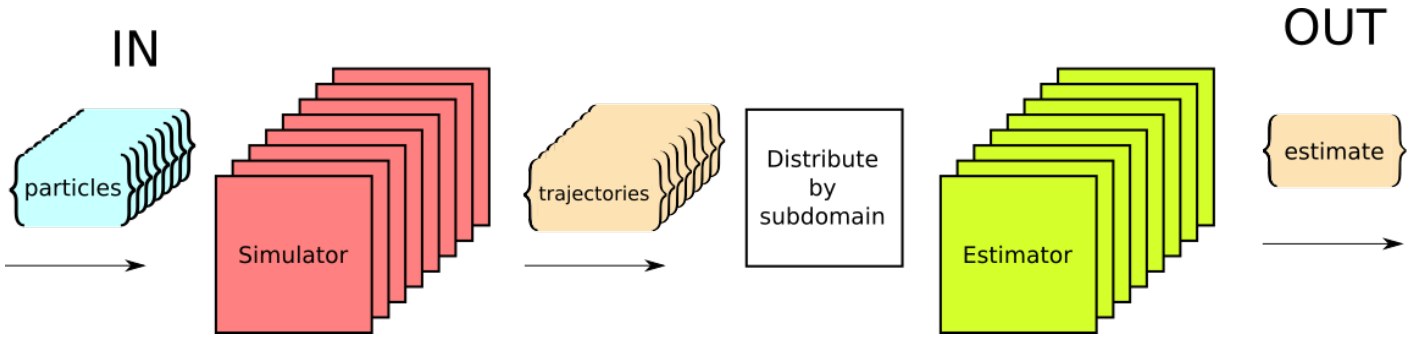
## Equivalent to current design of EIRENE+MPI




+ No shared mutable state → no synchronization until reduction
- Memory footprint scales linearly with number of processes
- Whole grid ($\times n\_threads$) must fit on one node

STATUS: DONE

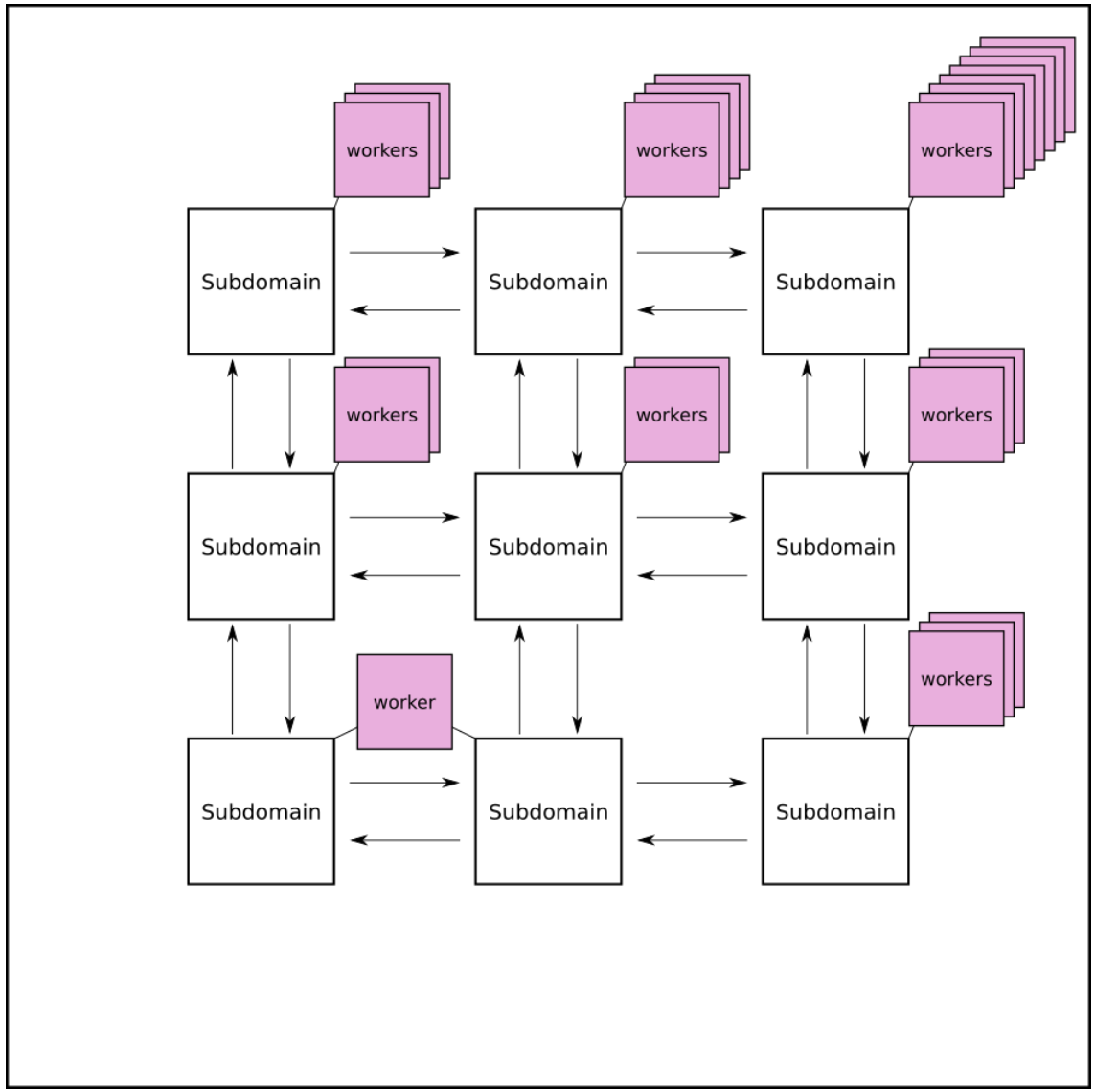
# Parallel designs: MPI pipeline




+ Shared mem simulation $\rightarrow$ constant memory footprint
+ Domain decomp estimation $\rightarrow$ private grids, but memory-efficient
- Whole grid must fit on one node

STATUS: DONE, BUT REIMPLEMENTING WITH Charon
(more on Charon later)

# Parallel designs: Domain-decomposed simulation grid



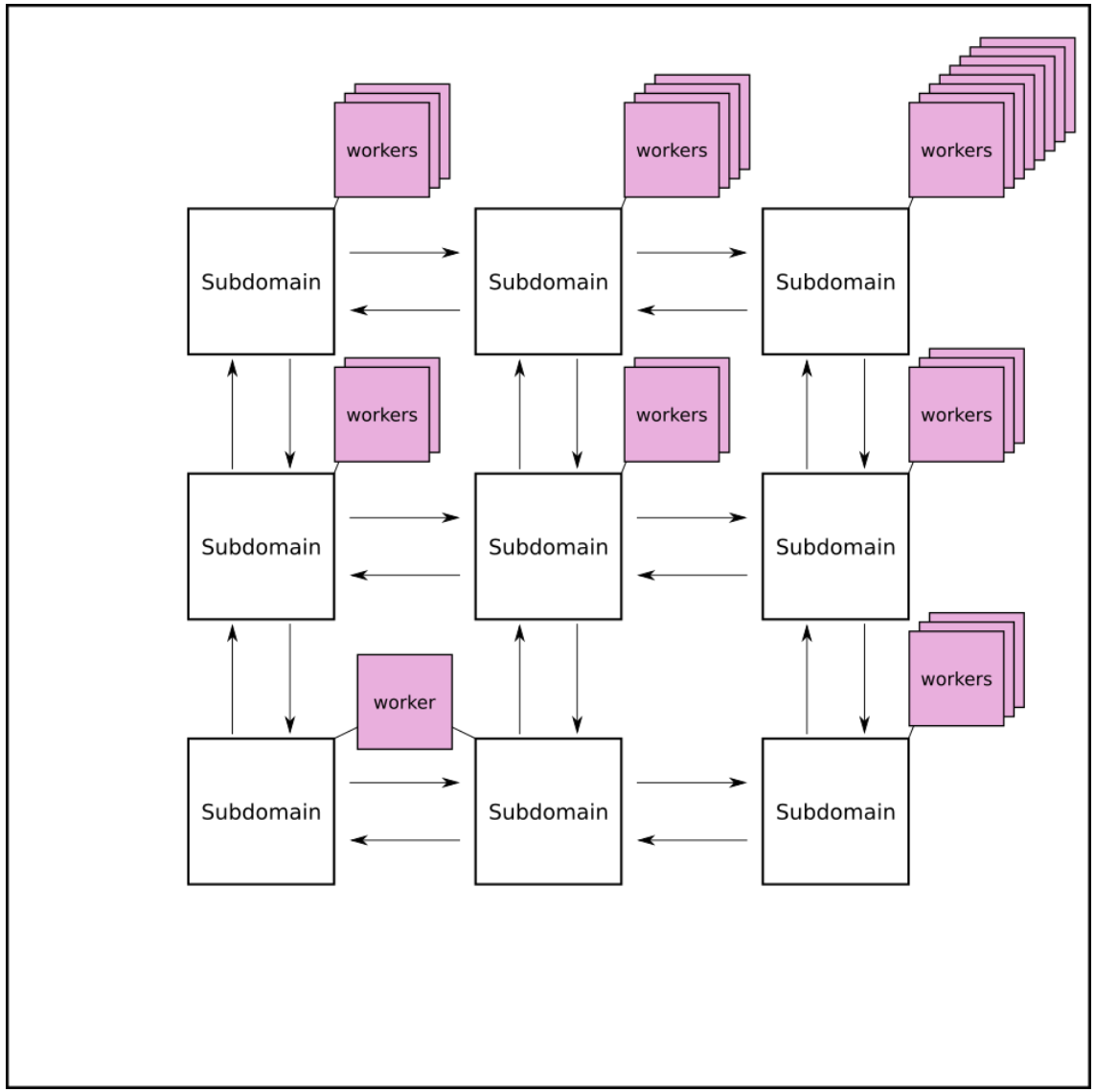
# Parallel designs: Domain-decomposed simulation grid

+ Domain decomp $\rightarrow$ memory use scales
+ Scale no longer limited to sim grid fitting on one node, can go multinode
+ Spatial load balancing now possible
- Load balancing is complicated

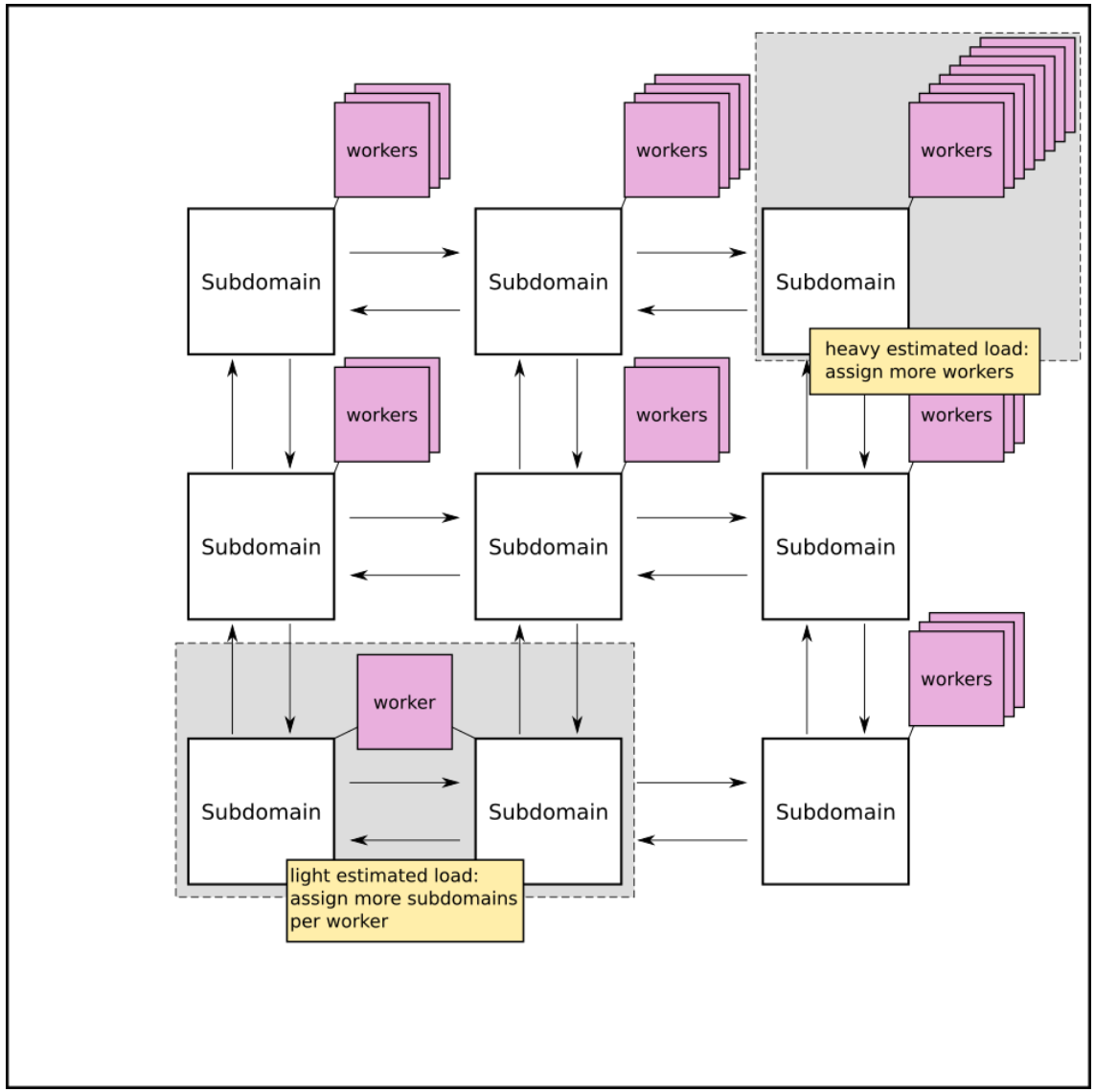
STATUS: DESIGN STAGE
DESIGN MOSTLY DONE

# Domain-decomposed particle tracing

# Domain-decomposed simulation: closer look



# Domain-decomposed simulation: load balancing idea



# Domain-decomposed simulation: load balancing idea

## Idea for load estimates

Run a low-resolution simulation (e.g. one cell = one subdomain), use produced density estimate as load estimate.

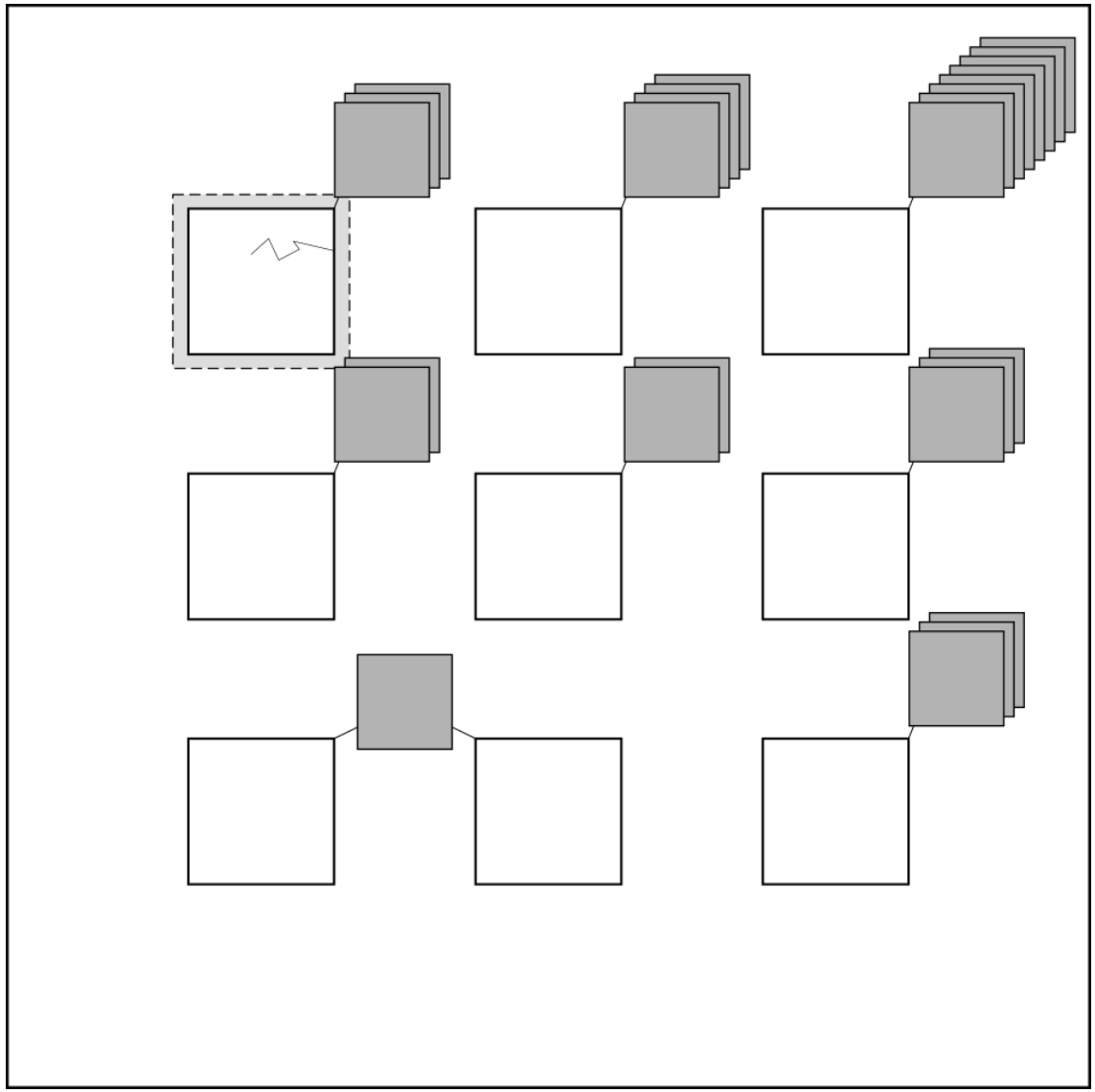
Bonus: because serial simulation is fully deterministic, we *could* run this in each process without communication.

# Domain-decomposed simulation: sticky particles

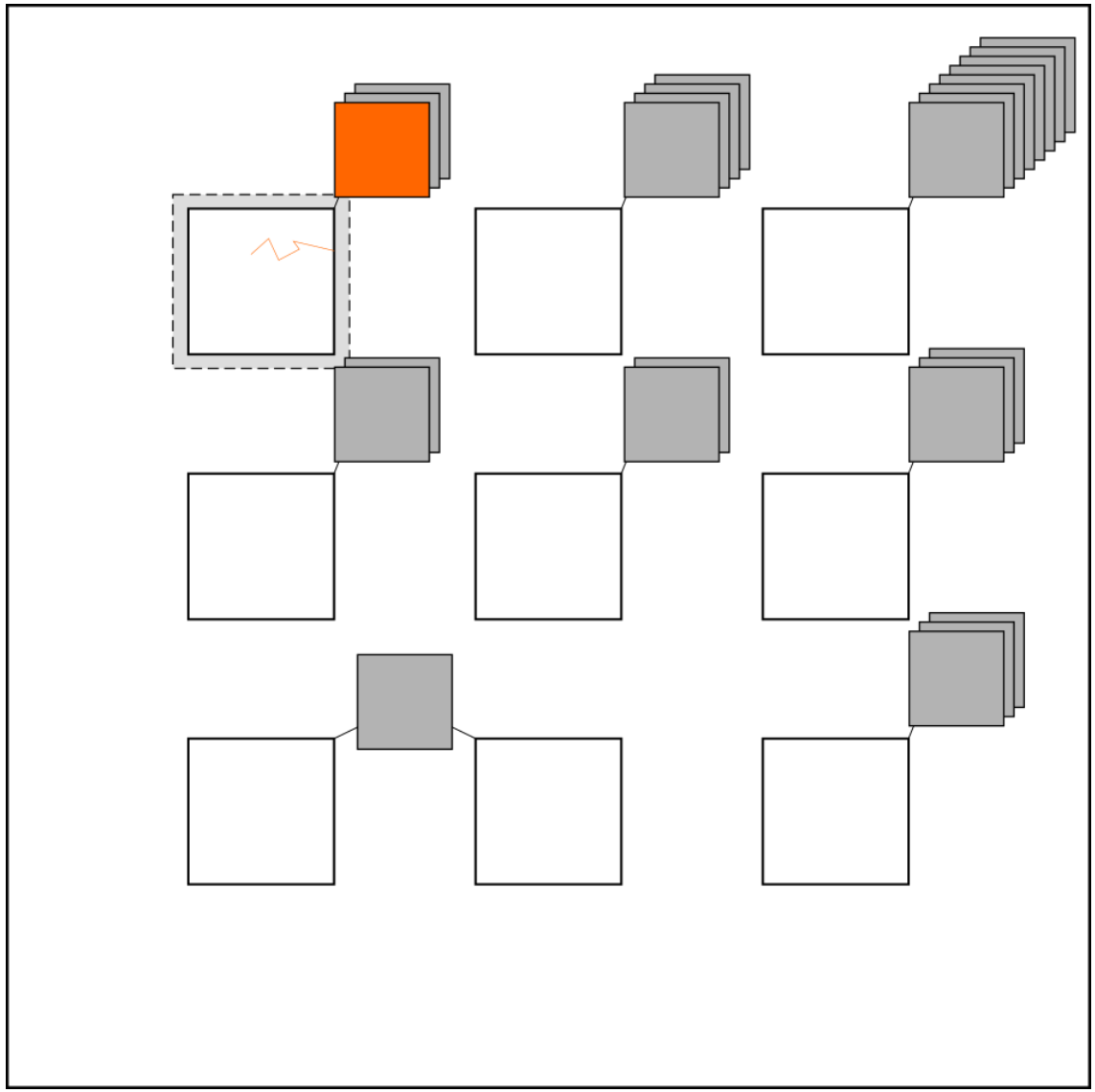
Because variance has to be counted per particle history,
the particles are *sticky* in each subdomain
— the same worker must process them each time.

Let's do a thought experiment to see what constraints follow.

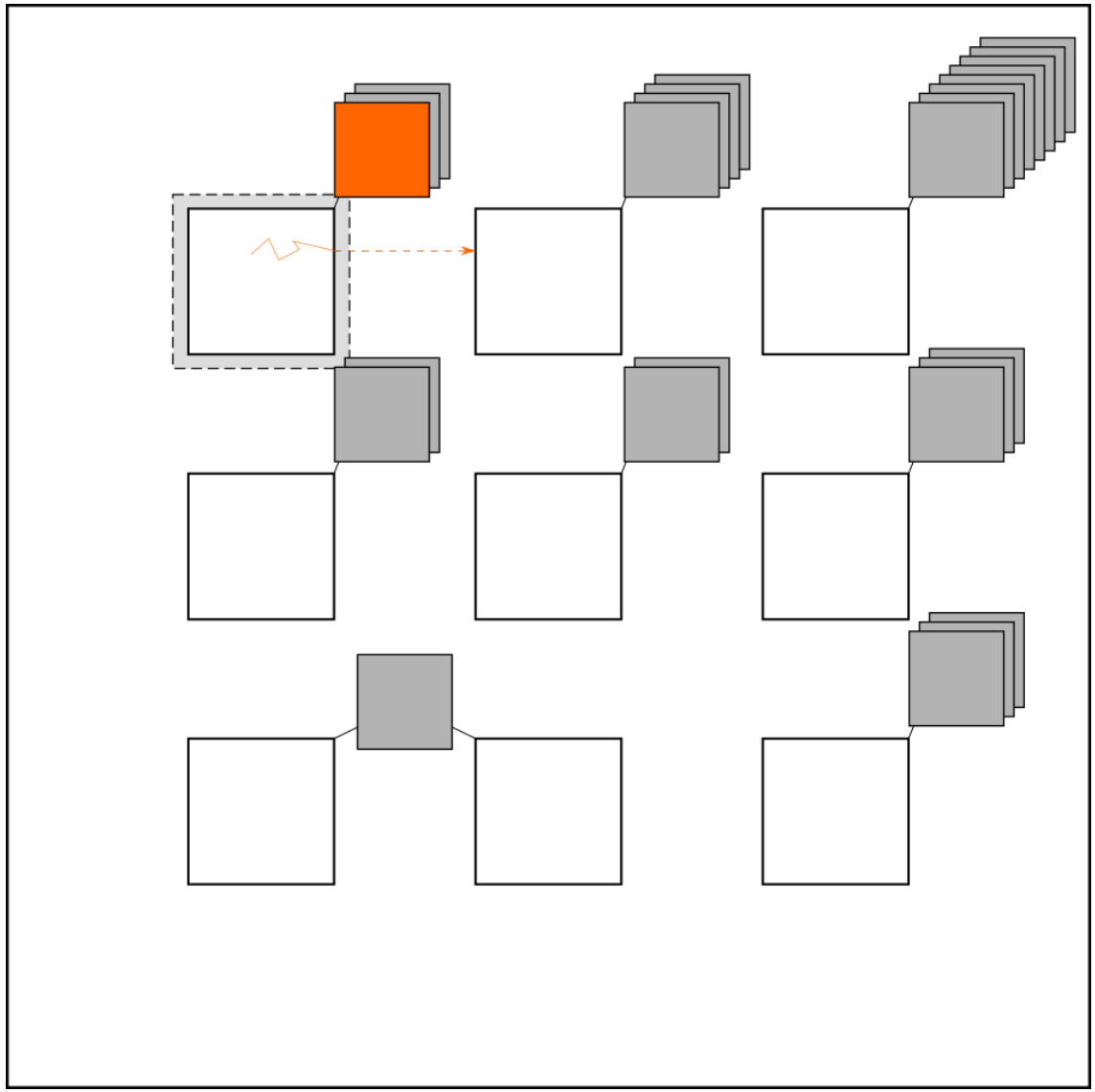
# Domain-decomposed simulation: example



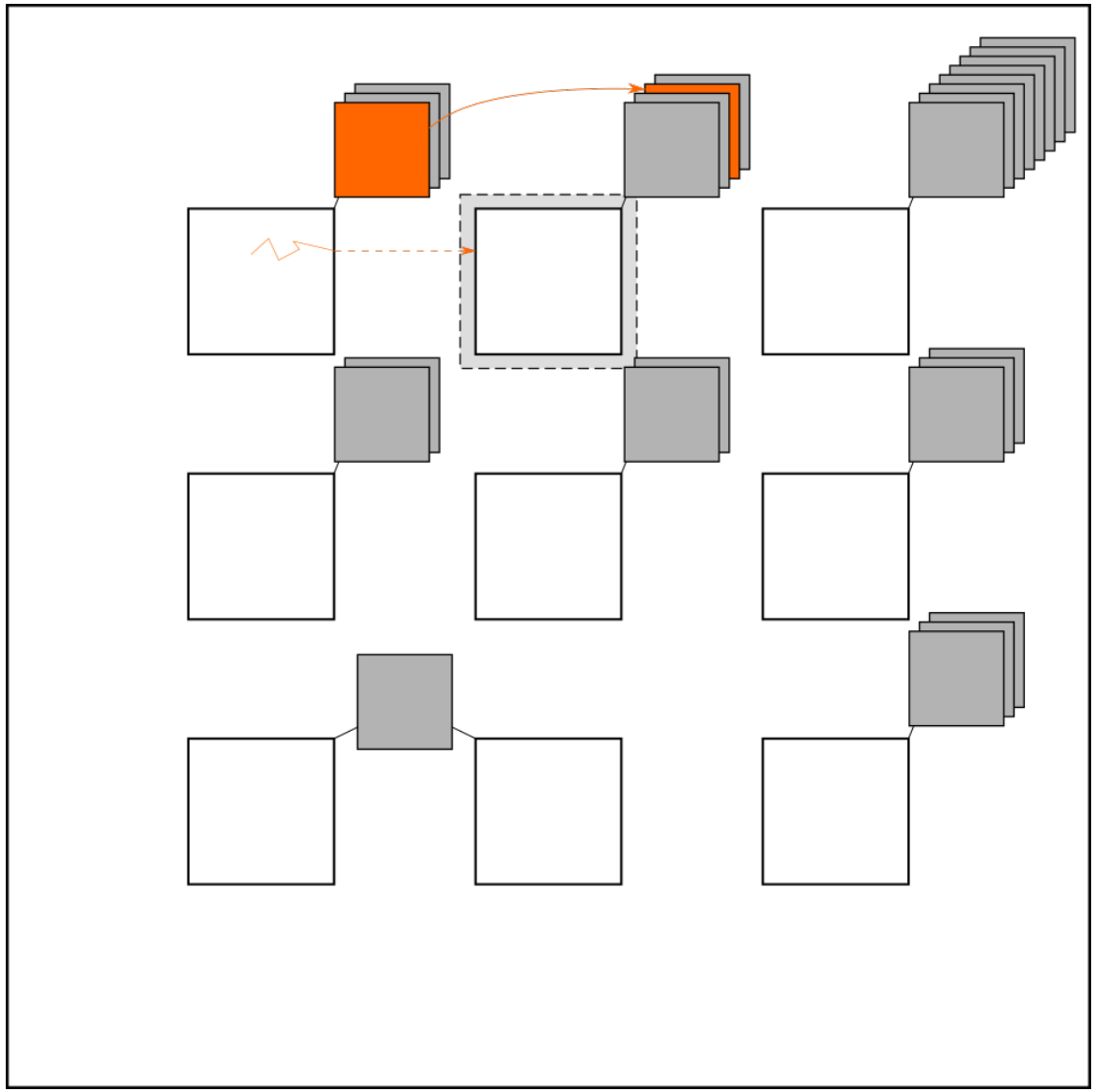
# Domain-decomposed simulation: example



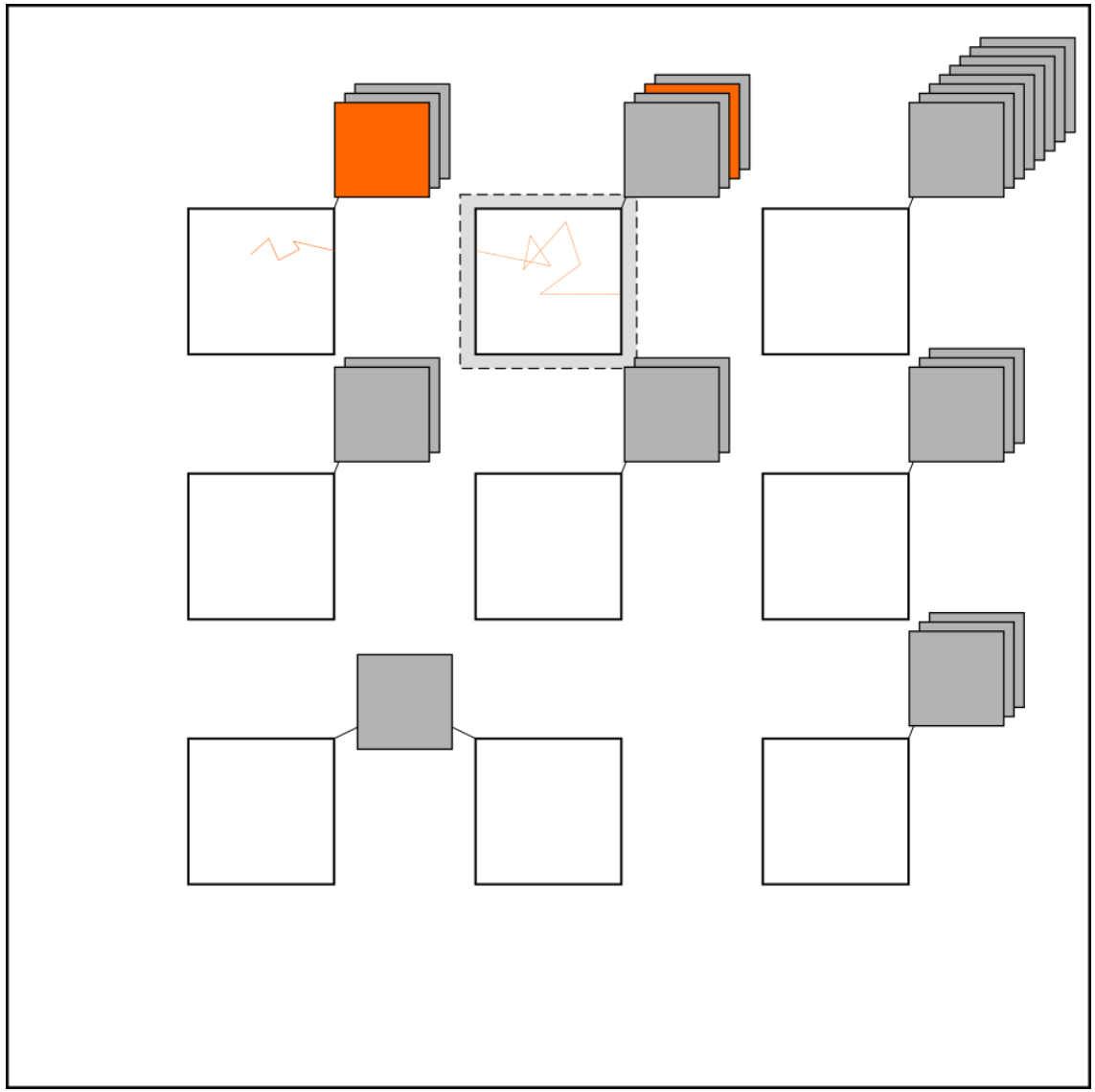
# Domain-decomposed simulation: example



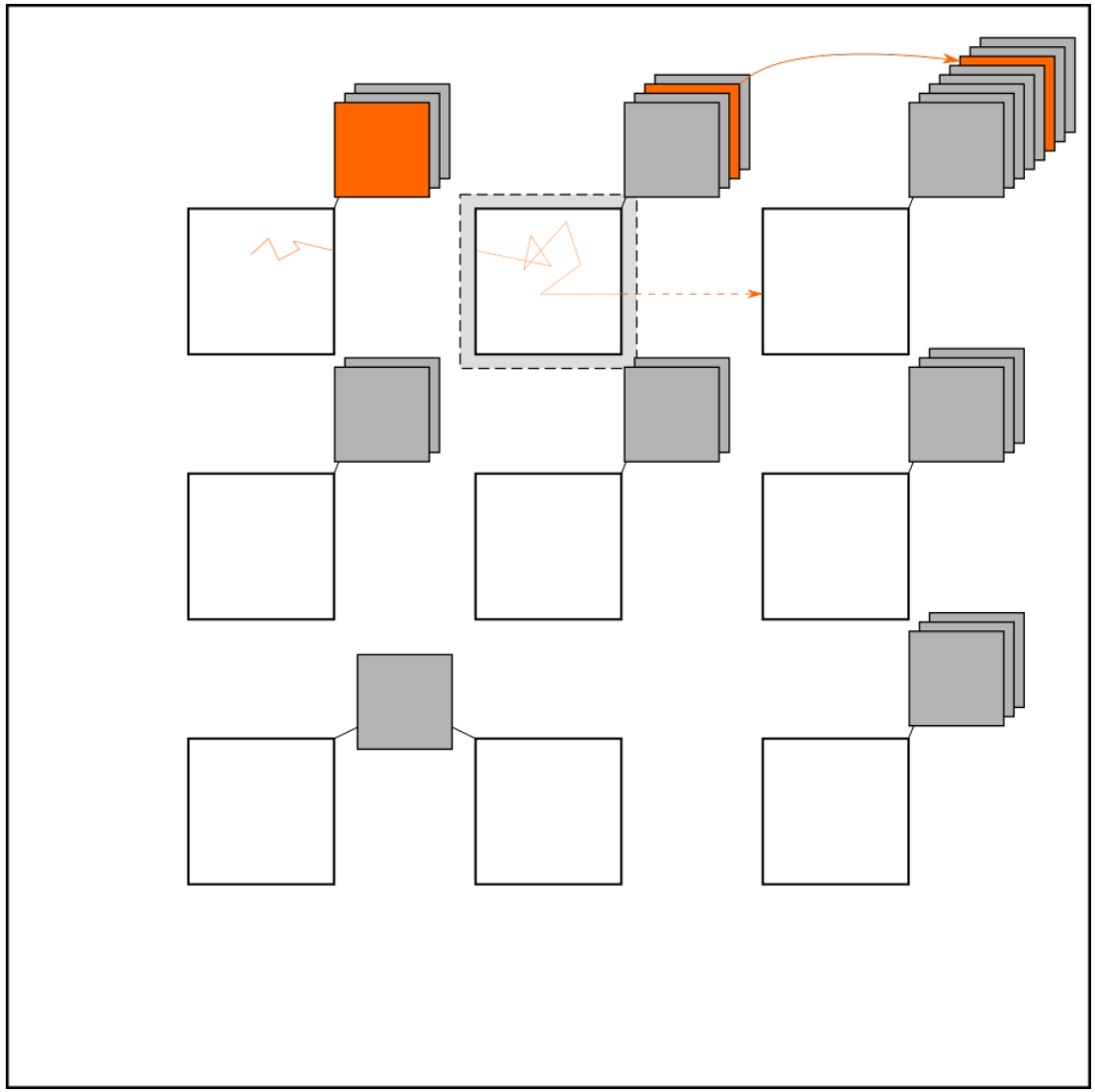
# Domain-decomposed simulation: example



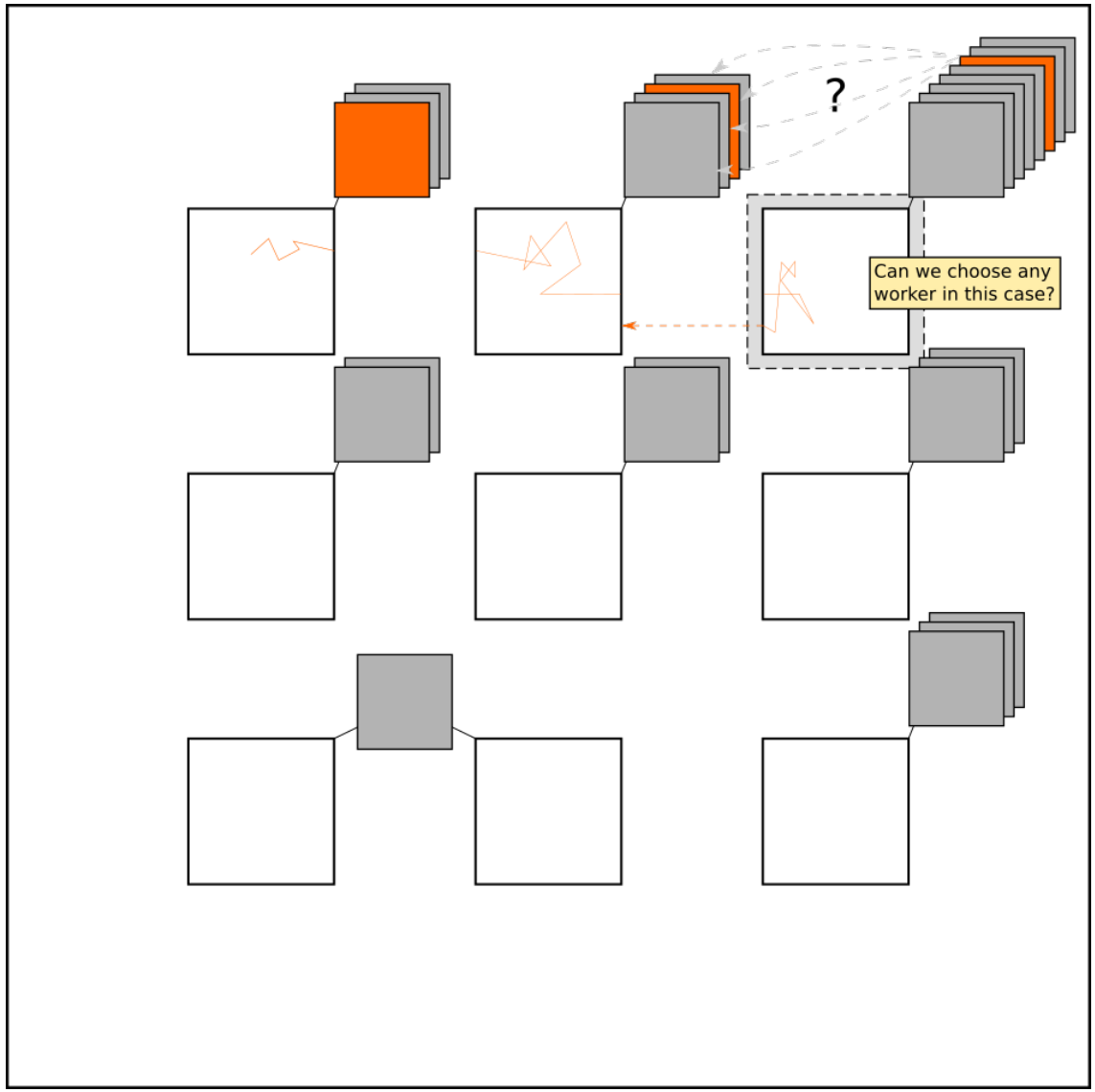
# Domain-decomposed simulation: example



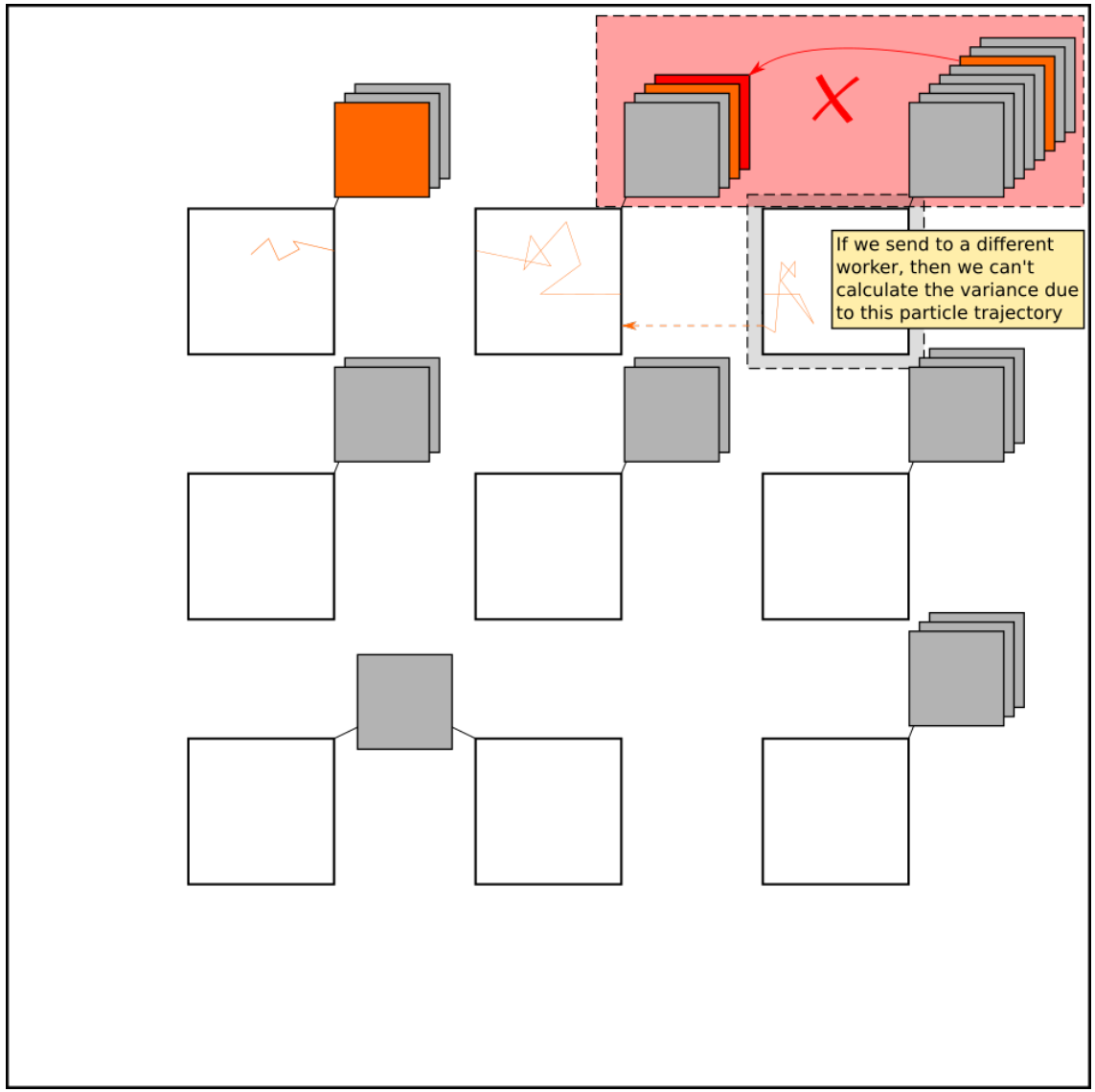
# Domain-decomposed simulation: example



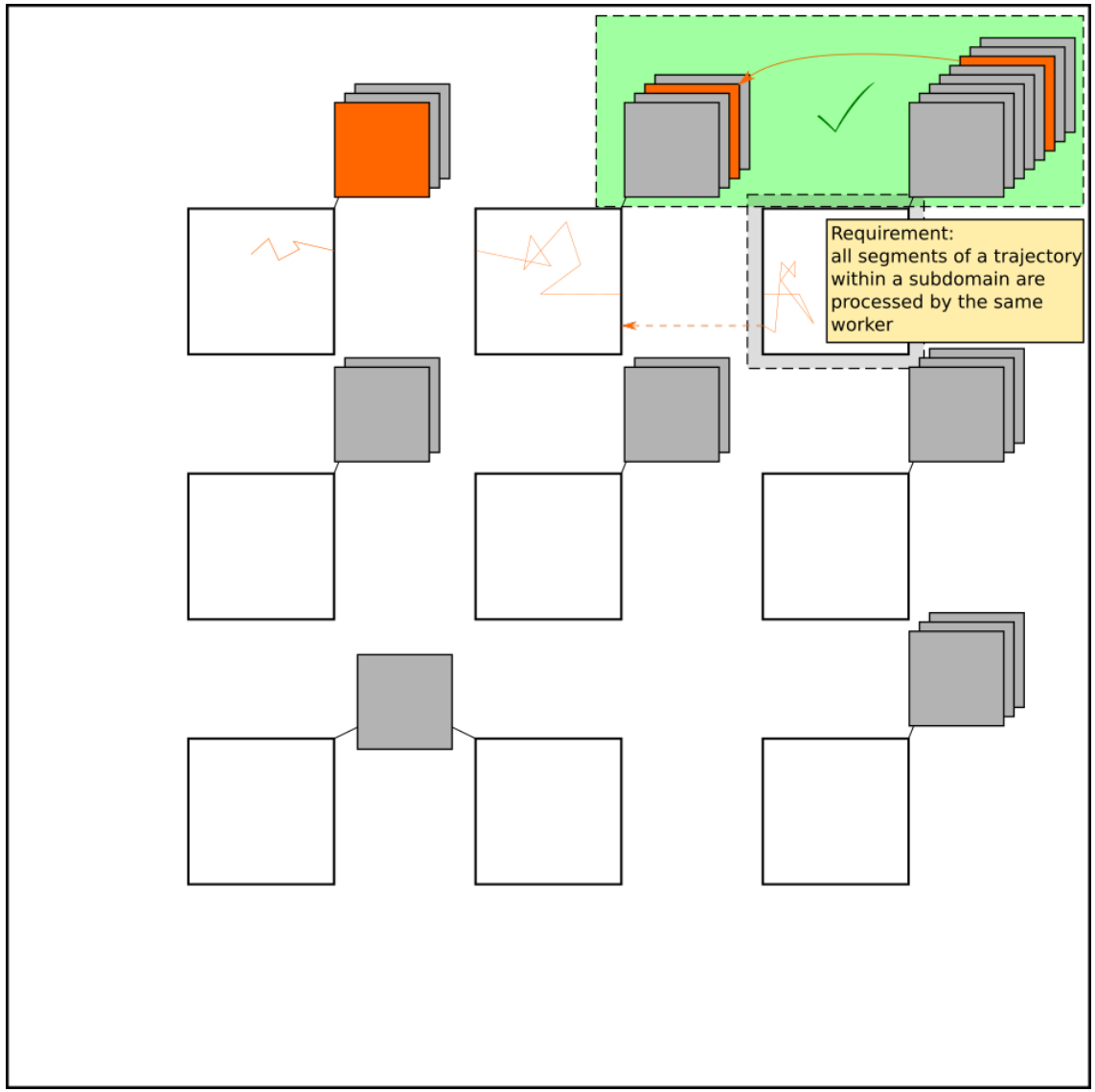
# Domain-decomposed simulation: example



# Domain-decomposed simulation: example



# Domain-decomposed simulation: example

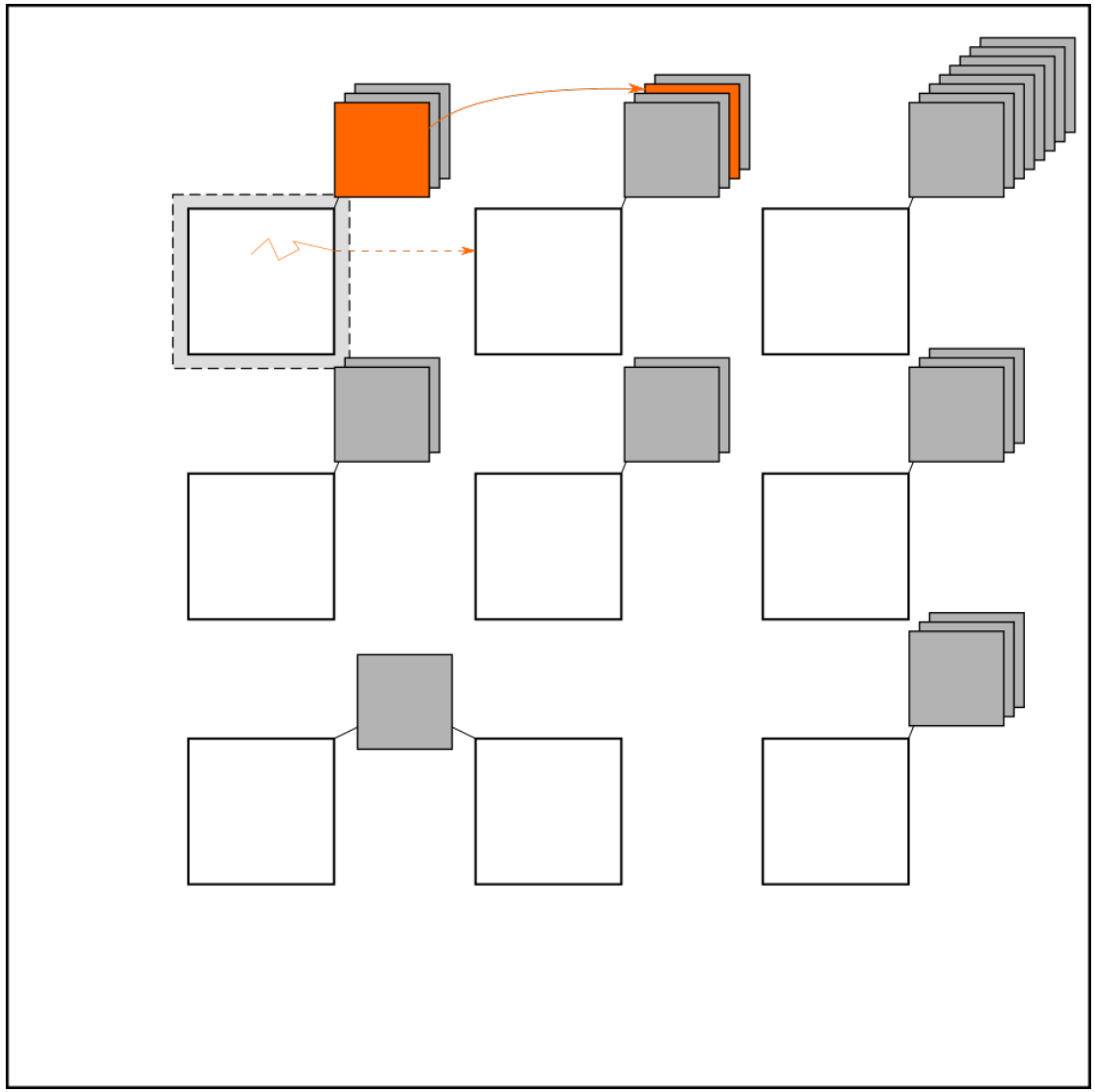


# Domain-decomposed simulation: sticky particles

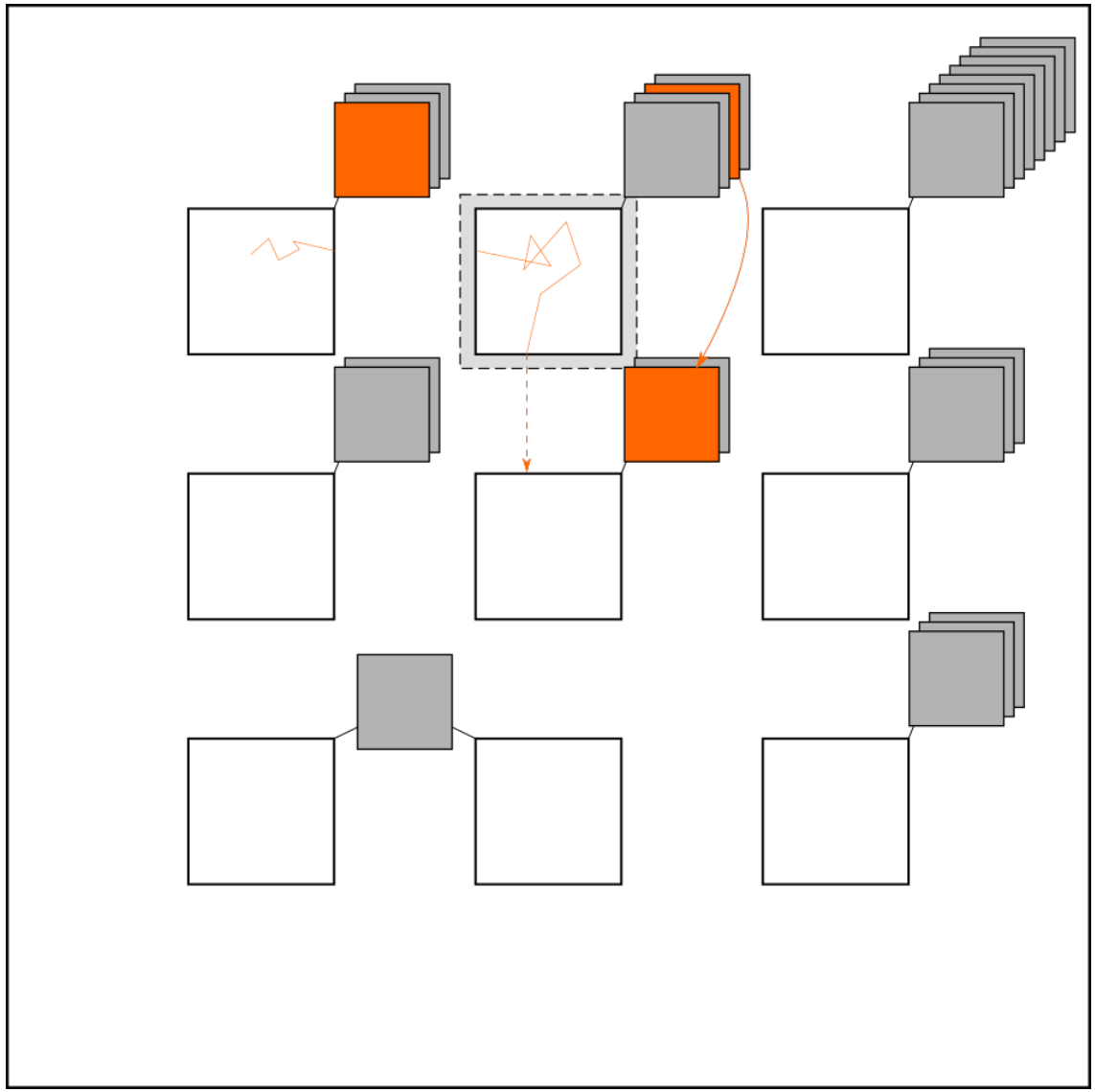
In this case, the worker could have just sent the particle stream back from whence it came.

Let's see where this is not the case.

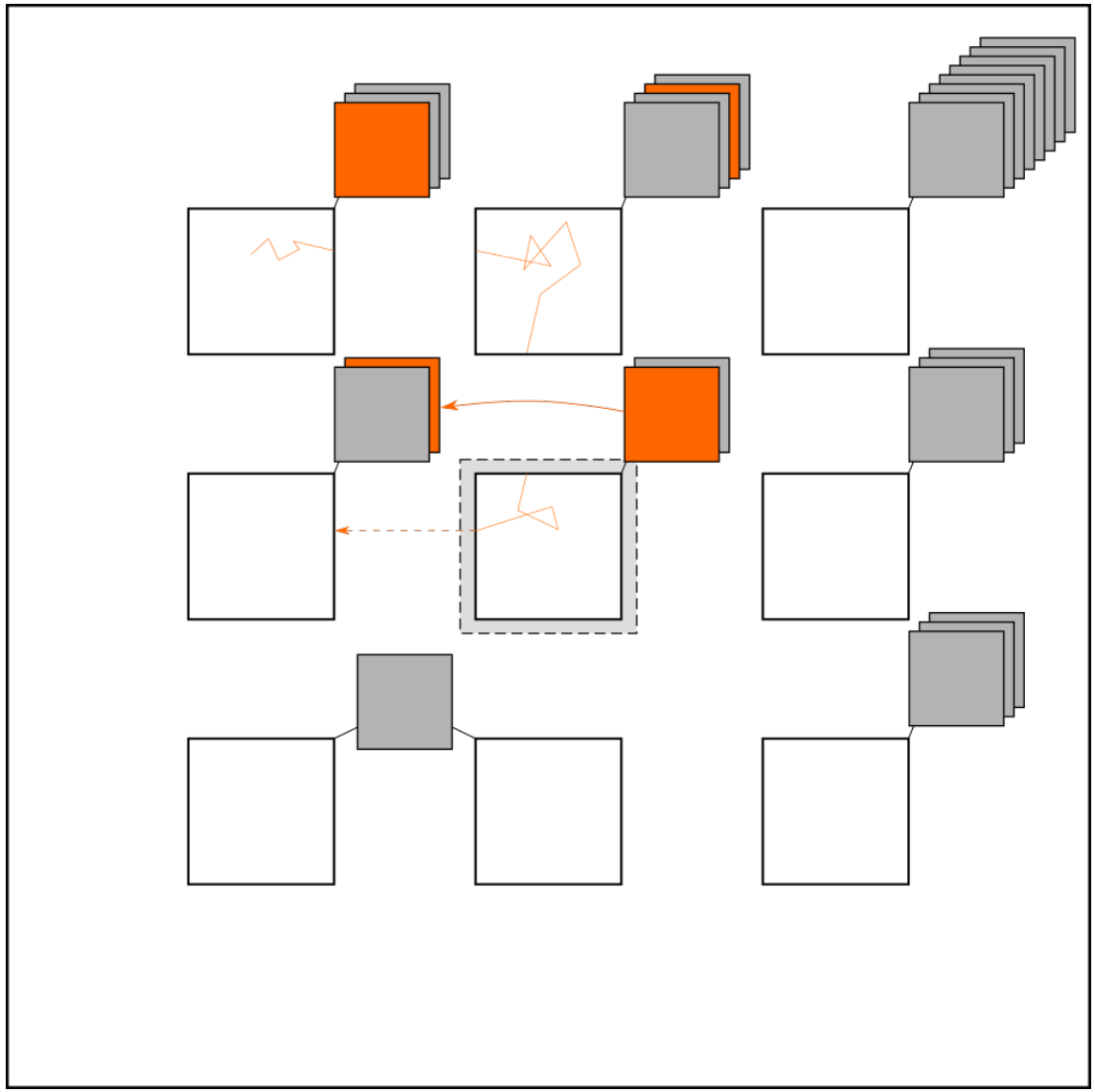
# Domain-decomposed simulation: example 2



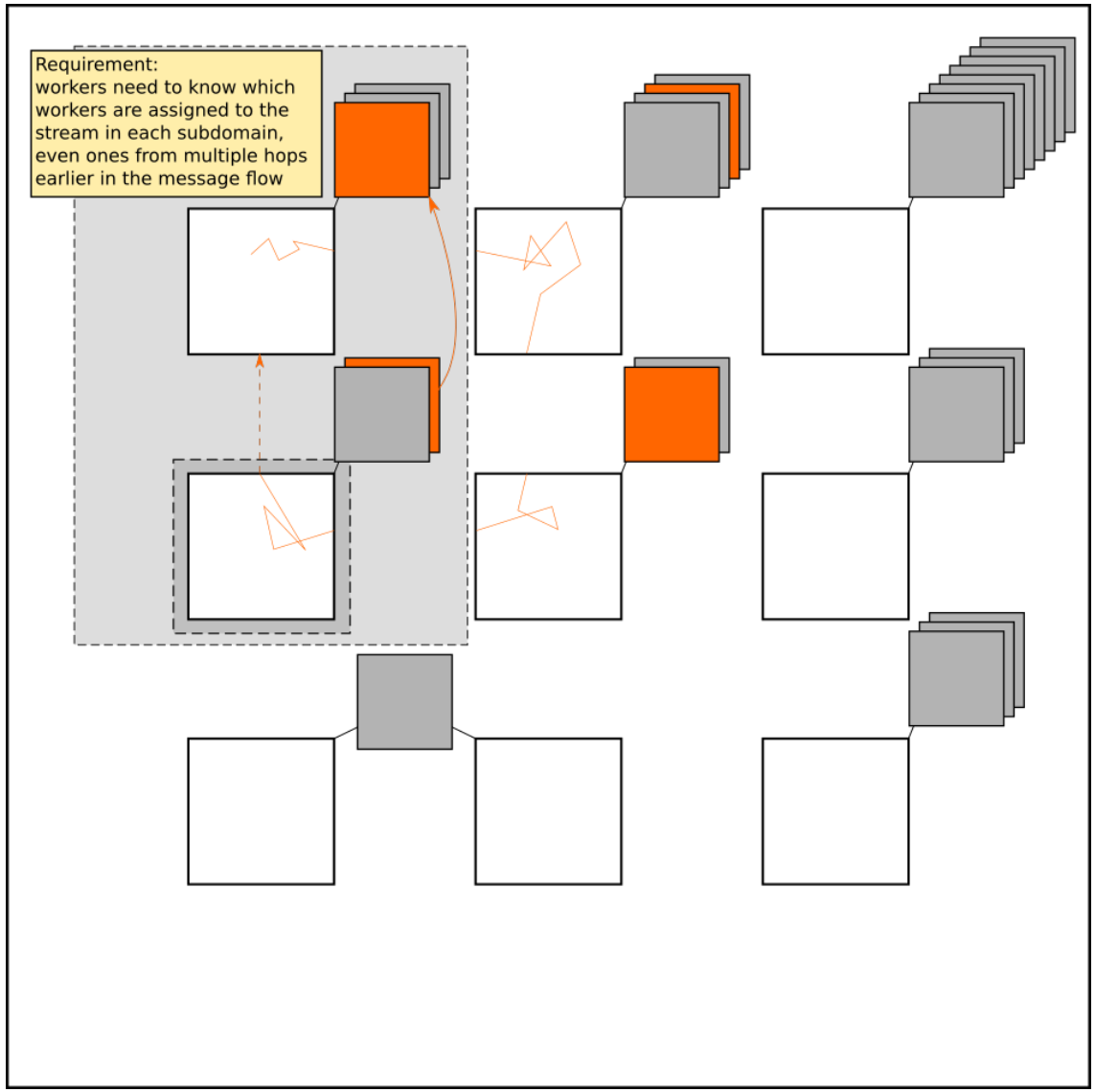
# Domain-decomposed simulation: example 2



# Domain-decomposed simulation: example 2



# Domain-decomposed simulation: example 2



# Domain-decomposed simulation: sticky particles

## Naive solutions

Store list of assigned workers in each visited subdomain

- not scalable (1000-10000 subdomains = kBs of overhead)

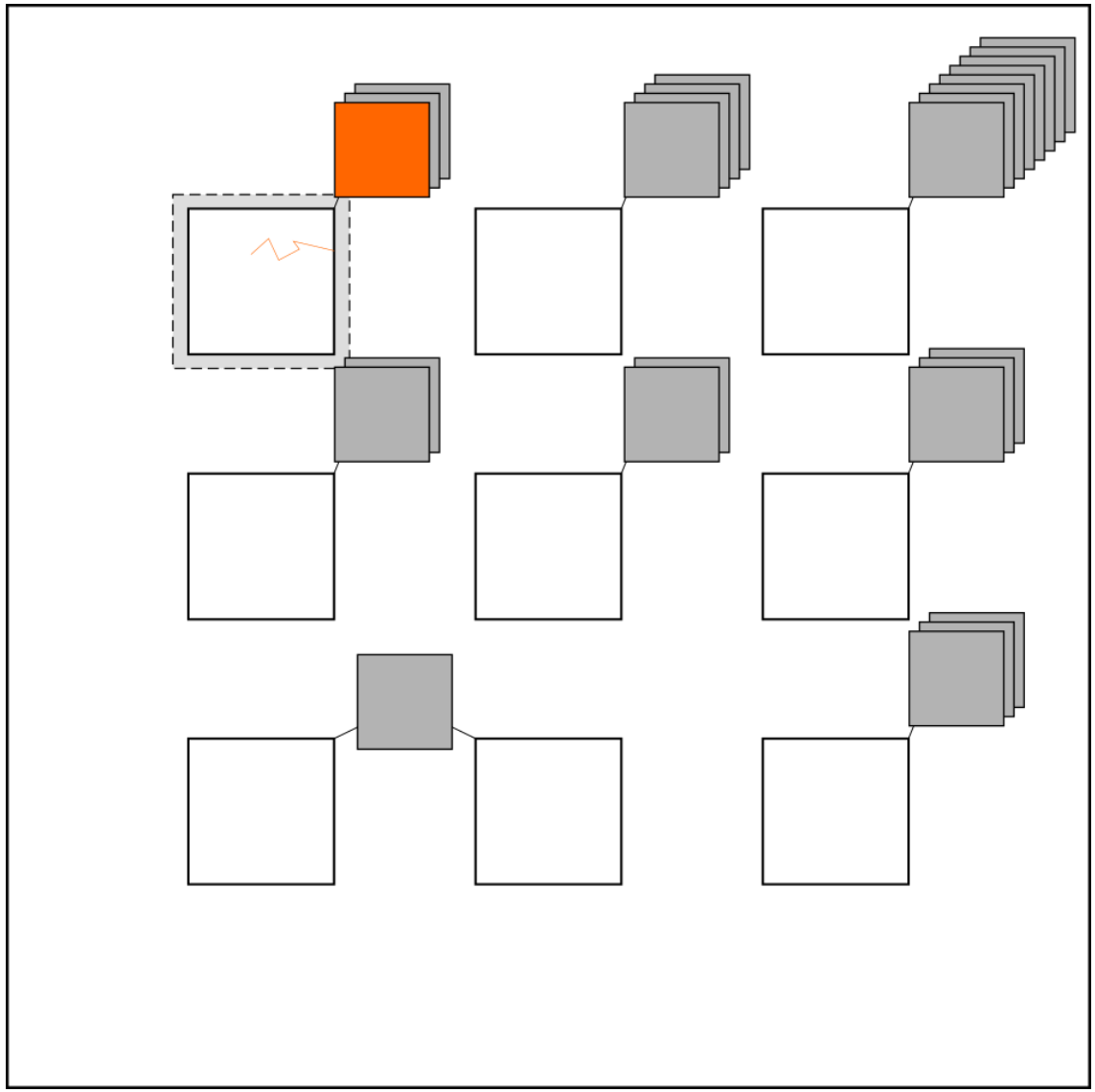
Index possible combinations of workers

- not scalable, worst case $n\_workers/2$ bits per message

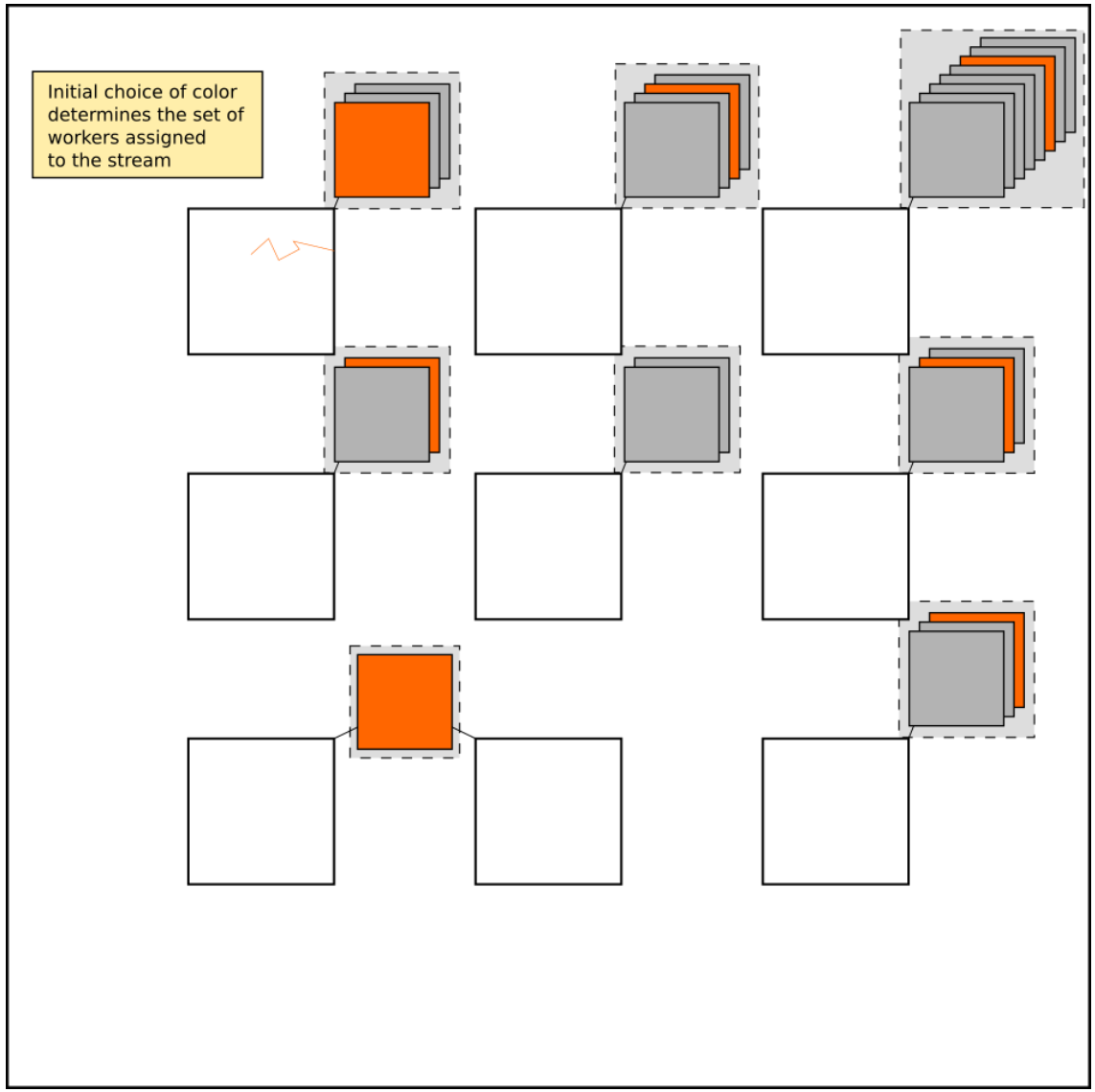
## Minimal overhead solution

Define a function that determines the worker from the particle id, and a binary choice. Then the initial choice of worker selects a worker in each subdomain.

# Domain-decomposed simulation: sticky particles



# Domain-decomposed simulation: sticky particles



# Domain-decomposed simulation: hashing

Choose some hash function $h$

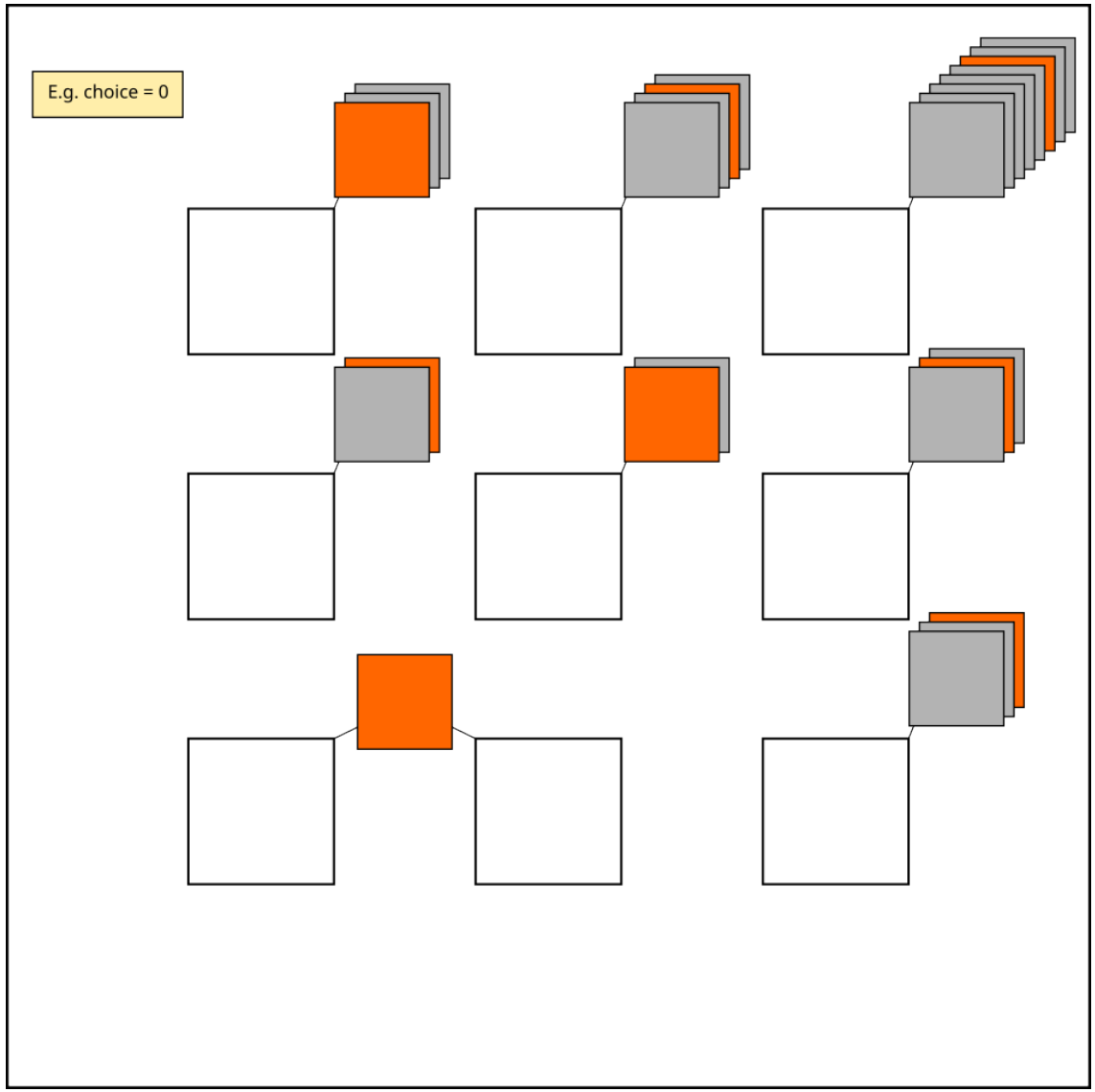
$$h : (particle\_id, choice) \to \mathbb{Z}$$

$$worker\ index = h(particle\_id, choice)\%workers\ in\ subdomain$$

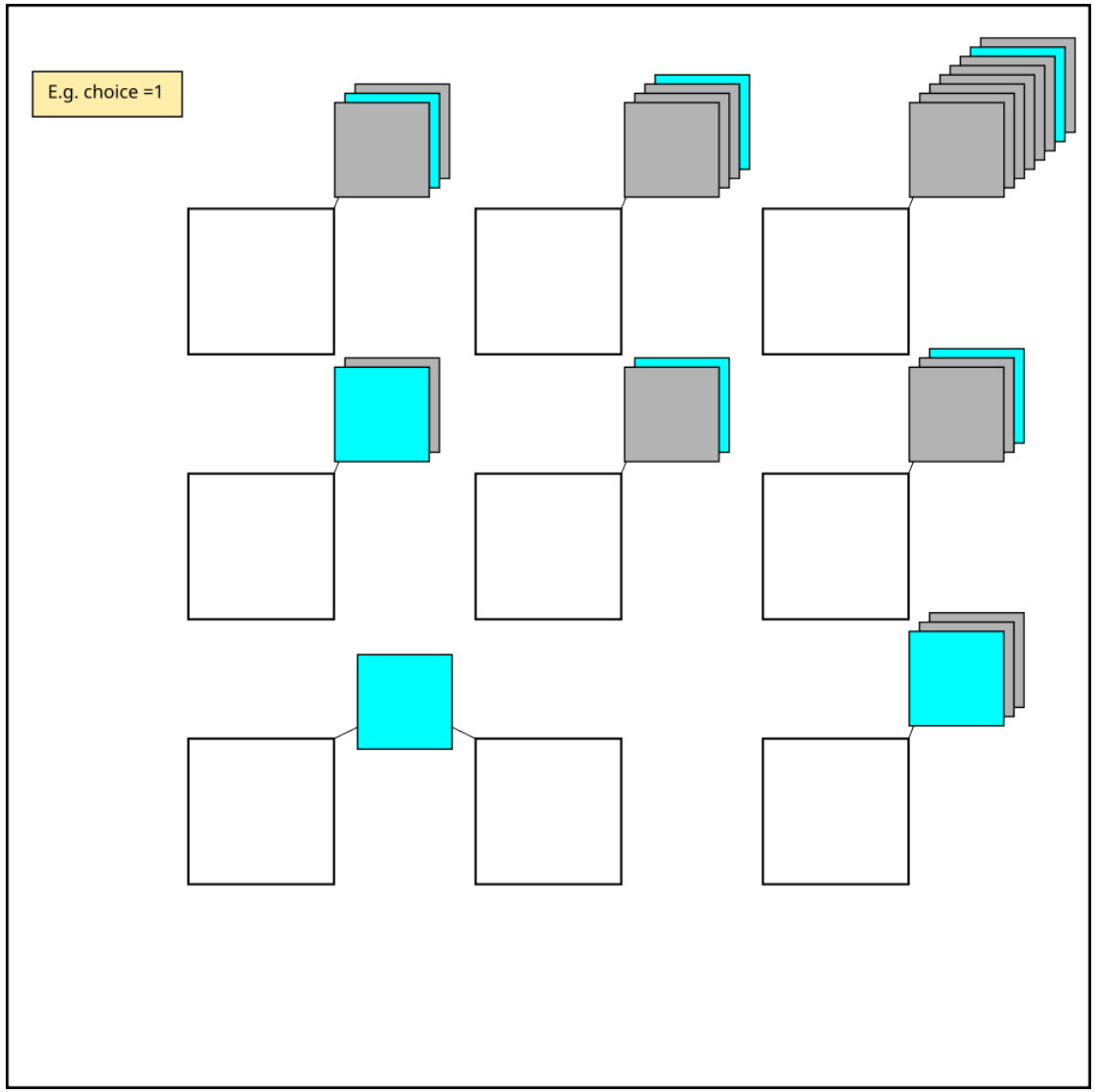
The additional parameter choice can be a single bit.
1-bit of overhead! We can encode that in an MPI tag.

An illustrated example follows.

# Domain-decomposed simulation: hashing



# Domain-decomposed simulation: hashing



# Domain-decomposed simulation: hash function

Extensive literature on hash functions, many options to choose from:

- Linear congruential generators (LCGs)
- MurmurHash
- Xorshift
- PCG
- etc.

Easy to make the choice of hash function a configurable parameter. Can make the decision later after benchmarking, instead of agonizing over the mathematical properties of each.

"When in doubt, use brute force."

- Ken Thompson

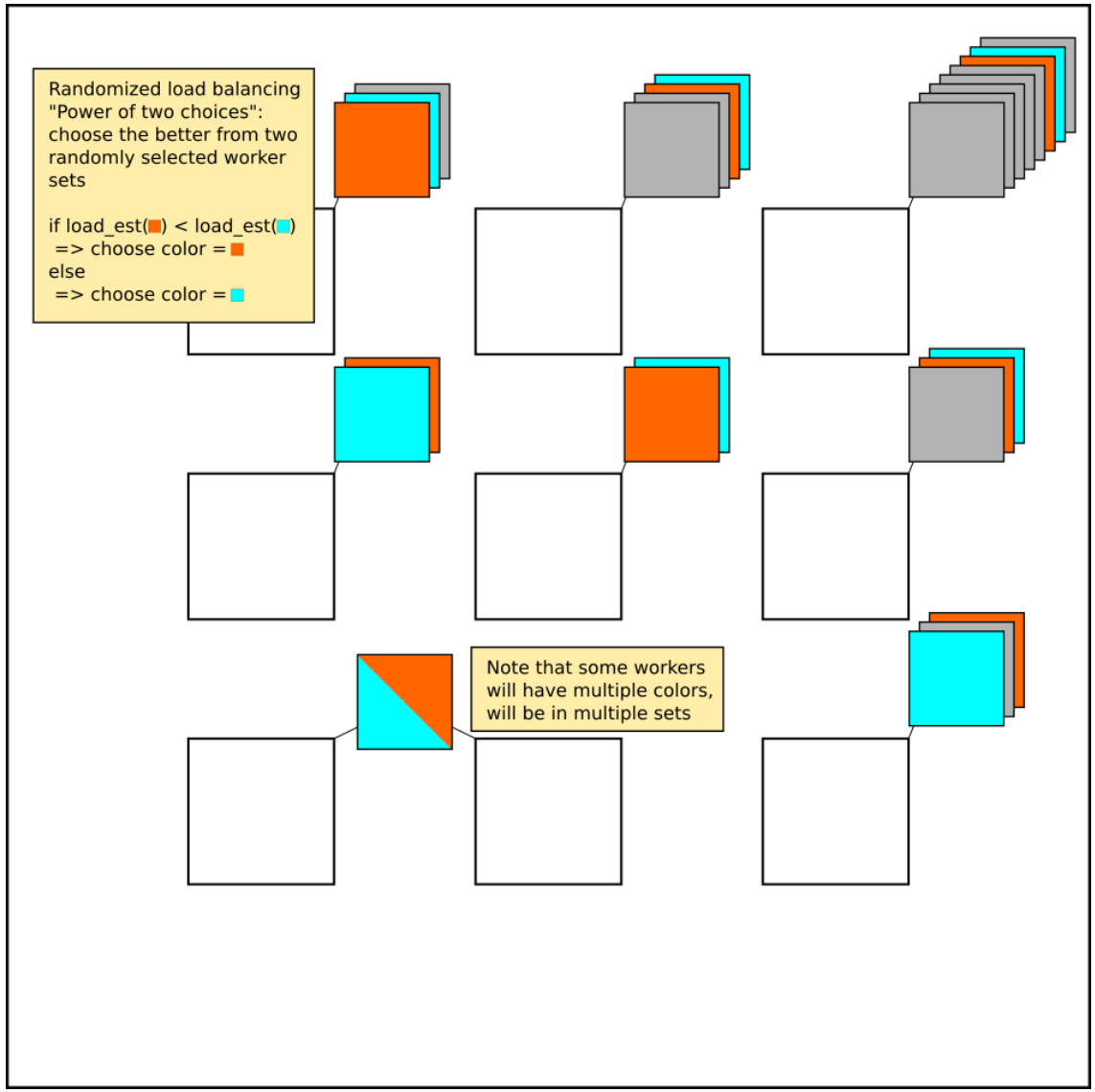
# Domain-decomposed simulation: power of two choices

The *choice* bit enables a dynamic load-balancing policy known as *randomized load balancing with power of two choices*.

Idea: randomly sample two servers to send a request to, select the one that has a lower estimated load

In our case, the sample is determined by the two hashes, and this is one design consideration for the hash functions, how well they distribute to two sufficiently uncorrelated sets.

# Domain-decomposed simulation: dynamic load balancing



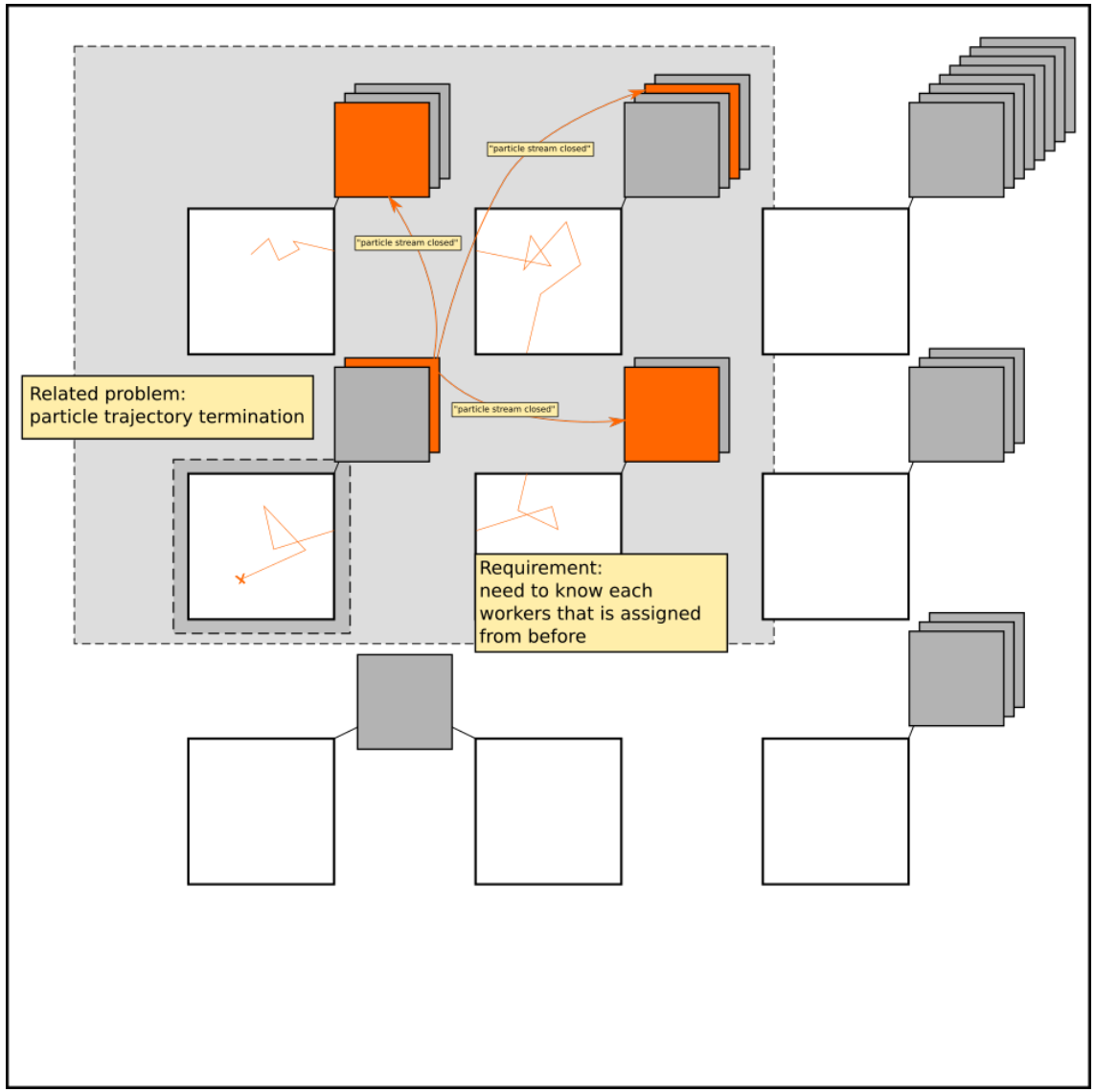
# Domain-decomposed simulation: dynamic load estimate

The only thing that remains is to choose this load estimate, and define how to distribute them back to the workers that generate particles, where the load balancing decisions are made.

Two simple solutions, both dependent on receiving particle termination messages immediately.

1. Use average time to process particle as load estimate
2. Use number of active particle streams as load estimate

# Domain-decomposed simulation: trajectory termination



# Domain-decomposed simulation: trajectory termination

Termination possible with one message *iff* we know which workers to send to in each subdomain. But we may send too many message with this method, the particle may not have visited each subdomain.

Alternative methods:

- Flood termination message back through all senders
- Global synchronization after N particles

# Domain-decomposed simulation: complexity

There is a lot of complexity involved in this solution, and solving it using raw MPI calls is cumbersome. A lot of the subproblems we need to solve are also quite generic.

$\rightarrow$ I've decided to create a separate library to handle the communication: Charon

# Charon concepts mapped to Eiron

## Channel

Eiron: All messages related to a subdomain are in one channel

## Streams

Eiron: All messages related to a particle trajectory form a stream

## Multichannel

Eiron: A multichannel allows a worker to process many subdomains

# Charon concepts: definitions

## Channel

Producers send messages to consumers through a channel

- Different channels are used for different kinds of data
- Channel consumers can process data specific to the channel

## Multichannel

A set of channels through which:

- Producers send messages with channel addressing
- Consumers receive messages from any of these channels

## Streams

A sequence of messages forming a meaningful whole.

- In each channel, between one producer and one consumer

# Charon: Roadmap

- ✓ Futures: Asynchronous MPI handles, wraps buffer+request
- ✓ Future pools: concurrent communication requests
- ✓ Variable sized messages
- ✓ Streams
- ✓ Distributed channel creation, producer-consumer assignment
- ✓ Multichannel base implementation
- ⇨ Replayable message traces for testing, debugging, profiling, and benchmarking
  - Multichannel: cross-channel stream lifecycle management
  - Multichannel: sticky streams

# Eiron: Roadmap

## Current project

- ➪ Get tracing and multichannels in Charon
  - Replace Eiron's bespoke MPI communication API with Charon
  - Reimplement MPI pipelines
  - Implement domain-decomposed grids
  - Benchmark approaches and analyze results
  - Publish initial results

## Backlog

- KDMC prototype
- Verification against EIRENE
- REQUIREMENTS FOR collision rate model
- REQUIREMENTS FOR geometry
- REQUIREMENTS FOR grid
- Trajectory representation comparison (depends on grid)

# Thank you

## Eiron

`https://version.helsinki.fi/lapposka/eiron`

## Charon

MPI channel library

`https://version.helsinki.fi/lapposka/charon`

## FFS

Portable CMake builds with on-demand dependency downloading

Used to build both Charon and Eiron

`https://version.helsinki.fi/lapposka/ffs`

## docker-devenv

Portable dev environments, spin up in any directory

Contains development dependencies for Charon and Eiron

`https://version.helsinki.fi/tools/docker-devenv`

# Appendix: Charon futures

A future is a handle to a value which will become available at some point. MPI has *requests*, which allow us to do what futures do, but with vanilla MPI we have to keep track of the memory buffer that the request is tied to, and the API is a little heavy. So Charon implements futures.

```
charon::mpi_future<int> fut;
fut.irecv(...);

if (fut.test()){
   auto status = fut.status(); //Wraps MPI_Status
   int message = fut.buffer;
   fut.reset();
   //Process message and status
   ...
} else {
   //do something else
   ...
}
```

# Appendix, Charon future pools

MPI will have to wait until we create an asynchronous request to copy data to the futures buffer. Future pools help. A future pool is a list of pre-allocated futures that we can assign to requests and complete one at a time.

```
charon::mpi_future_pool<int> futures;
auto *fut = futures.try_get();
if (fut != nullptr){
    fut.irecv(...);
}
...
fut = futures.test_any();
if(fut != nullptr){
    //process future
}
```

# Appendix, Charon channels

Channels have internal future pools for received messages, but they take a future as an input parameter for sent messages, so that the sender can check the status of the message request without asking the channel object.

```
//Producer view
charon::async_channel chan(...);
auto *fut = futures.try_get();
create_message(fut.buffer);
chan.send(fut, stream_tag);
...
//Consumer view
charon::async_channel chan(...);
auto *fut = chan.recv();
//OR
auto *fut = chan.try_recv();
```